AFRL-IF-RS-TR-1999-144
Final Technical Report
July 1999

SOFTWARE EVOLUTION THROUGH AUTOMATIC MONITORING

Computing Services Support Solutions

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. D931/D961

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-1999-144 has been reviewed and is approved for publication.

APPROVED: Deborah A Cerino
DEBORAH A. CERINO
Project Engineer

FOR THE DIRECTOR: Northrup Fowler III
NORTHRUP FOWLER, III, Technical Advisor
Information Technology Division
Information Directorate

If your address has changed or if you wish to be removed from the Air Force Research Laboratory Rome Research Site mailing list, or if the addressee is no longer employed by your organization, please notify AFRL/IFTD, 525 Brooks Rd, Rome, NY 13441-4505. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

SOFTWARE EVOLUTION THROUGH AUTOMATIC MONITORING

K. Narayanaswamy

Contractor: Computing Services Support Solutions
Contract Number: F30602-98-C-0270
Effective Date of Contract: 30 August 1996
Contract Expiration Date: 31 January 1999
Short Title of Work: Software Evolution Through Automatic Monitoring

Period of Work Covered: Aug 96 - Jan 99

Principal Investigator: K. Narayanaswamy
Phone: (213) 299-3136
AFRL Project Engineer: Deborah A. Cerino
Phone: (315) 330-1445

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by Deborah A. Cerino, AFRL/IFTD, 525 Brooks Rd, Rome, NY.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	Jul 99	Final Aug 96 - Jan 99

4. TITLE AND SUBTITLE

SOFTWARE EVOLUTION THROUGH AUTOMATIC MONITORING

5. FUNDING NUMBERS

C - F30602-96-C-0270
PE - 63728F
PR - D931
TA - 01
WU - 01

6. AUTHOR(S)

K. Narayanaswamy

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Computing Services Support Solutions
5777 West Century Blvd, Suite 1230
Los Angeles, CA 90045-5600

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington, VA 22203-1714

AFRL/IFTD
525 Brooks Rd
Rome, NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

AFRL-IF-RS-TR-1999-144

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: Deborah A. Cerino, IFTD, 315-330-1445

12a. DISTRIBUTION AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This report summarizes the research accomplished and the resulting sophisticated software monitoring infrastructure that was developed during the contract period. Monitoring is a critical component in controlling any adaptable system. As part of this effort, a system called software monitoring service (SoMoS) was developed. SoMoS is essentially a data fusion service, collecting low level data, and combining it to produce results of interest to software developers (or any consumer). SoMoS accepts primitive events and delivers notification of derived events over a network and is therefore a useful service for network applications. This system has the ability to add monitors dynamically, an important requirement for active network monitoring. This service is applicable to a host of applications that require the ability to make decisions based on correlations between distributed events.

14. SUBJECT TERMS

Event monitoring, Event calculus

15. NUMBER OF PAGES

30

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Table of Contents

I Executive Overview

This document is the Final Technical Report for a research project entitled "Software Evolution through Automatic Monitoring" conducted under the Evolutionary Development of Complex Software program sponsored jointly by the Defense Advanced Research Projects Agency (DARPA), and the Air Force Research Laboratory (AFRL), Rome, New York.

The broad goals of this research project were:

a) To develop monitoring infrastructure for fielded software systems that would be both incremental and dynamic
b) To provide a more reactive, automated framework to notice when software systems could benefit from evolutionary changes and how

The underlying technical vision was that one could then monitor software systems in their operating environments for mismatches between expectations and reality.

In addition, it was deemed important to show that monitoring could be added to "low-end" programming environments such as Unix or Windows (i.e., bare operating systems), as well as to sophisticated programming environments that had more knowledge about the software artifacts that were being developed.

All the above goals have largely been achieved during this technology project, and this document briefly describes the following:

a) A brief chronological history of the project
b) Technical overview of the project
c) Descriptions of the key technologies developed and how one can use them
d) Lessons learned through this research project are included

In particular, the Software Monitoring Service, or ***SoMoS***, along with the Flea event specification language, are the direct and tangible results of this project. ***SoMoS*** is now a fully functioning product, licensed by Cs3 as a product to build monitoring applications. This document concludes with a brief account of the commercial successes of this project.

II Introduction and Background

This project was based upon a simple observation: once software systems are fielded, there is little about them that is directly communicated to the developers of the system. Yet, there is so much that developers can learn a lot about the effectiveness (or lack thereof) of their design decisions by watching how a system fares in its operating environment.

All software systems are built under implicit or explicit assumptions. If these assumptions are not valid in the operating environment, systems typically deteriorate in performance, or, in the worst case, can actually fail. We believed that this could be remedied with an elegantly simple idea: automatic monitoring…

Accordingly, the major thesis underlying this project was that software systems could be monitored in some manner that would provide valuable feedback about fielded systems to software developers, designers, and configurers. The idea was that such feedback would prove useful in knowing that a software system was in an environment where some of the assumptions underlying its design were not satisfied. Not only could such notifications prove valuable to fix performance and other problems, they could also identify opportunities for improvement and optimization.

Traditional monitoring (such as operating system or network monitoring) is far too low-level for the kind of monitoring needed for this project. Further, almost all monitoring regimes were static and required a-priori knowledge of all the key monitoring conditions. Typically, this is because monitoring conditions are hard-wired into the software system, and, therefore, require compilation. Therefore, we wished to separate the monitoring component of the system as a separate function from the system itself.

In this particular application, namely software monitoring, it is unreasonable that all the important monitoring conditions be known ahead of time. Thus, one of the important requirements for the monitoring framework for this project would be that the monitored conditions be changeable even after the software is fielded.

Thus, in pursuing the goal of automating the detection of when software systems needed to evolve, we essentially had to build our own monitoring service, which, ironically, became the largest research and commercializable contribution of this project.

III Goals of the Research

Having identified the basic need in the software field for sophisticated, generic, and dynamic monitoring technology, the major technical goals were as follows:

1. Develop a generic monitoring language in which one could state complex assumptions in terms of simpler environmental events.

2. Build the language to be both powerful and compilable, and with the ability to state new criteria dynamically and on the fly.

3. Provide for a notification architecture that would make such a monitoring service generally useful across a variety of platforms.

4. Integrate the monitoring service into generic operating systems such as Windows or Unix, but also into an advanced programming environment which maintains knowledge about the software being developed. The Rome Laboratory Knowledge-Based Software Assistant Advanced Development Model (or KBSA ADM) system was identified as the candidate going into this project.

IV Project History and Assessment

Project History

This project was envisioned as a two-year research effort. It started on September 1, 1996. In addition to the Principal Investigator, Dr. K. Narayanaswamy, the project included as senior scientists Drs. Martin Feather and Donald Cohen.

In the original conception of this work, we had assumed that UNIX and Windows would be the "normal" operating systems into which we would integrate monitoring. However, part of the research effort was to integrate monitoring into a true programming environment which could provide not only more knowledge about the software being developed, but perhaps a larger range of automated responses if and when problems were detected. It was thought that the Knowledge Based Software Assistant (KBSA) system Advanced Development Model (ADM) would be the system of choice here.

In Year I, we had developed the necessary constructs of the Flea language, and implemented the event specification translators. In addition, rudimentary socket-based interfaces to a generic monitoring service were defined. The KBSA/ADM integration was found to be infeasible because of lack of support for that system. However, the integration into UNIX/Windows was well under way. In July 1997, these features were demonstrated at the EDCS Demo Days extravaganza in Seattle. The technology was shown at the Cs3 booth, and also in the USC/ISI booth, where monitoring a la Flea was used in detecting architecture constraint violations.

In Year II, we consolidated the basic monitoring service, and built up tool infrastructure around it. For example, syntax directed editors, and other support tools were built. The resulting system was christened Software Monitoring Service or SoMoS. This is a functioning monitoring framework. The UNIX/Windows integration were fully completed. On the question of integration with a more advanced system, we began work on the Catalyst engineering environment with ModusOperandi of Indialantic Florida. A design was made, but not completed in this year. In July 1998, SoMoS was demonstrated once again at the EDCS Demo Days function, this time in Baltimore. SoMoS was also represented at the ModusOperandi booth in a Catalyst/SoMoS joint scenario.

The project exercised a no-cost contract extension to January 1999, beyond its nominal end of August 1998. This allowed us time to complete the Catalyst integration with ModusOperandi, and plan a final demonstration of the SoMoS/Catalyst integration at Cape Canaveral Air Station (CCAS) in January 1999.

Project Assessment

Each of the technical goals 1, 2, 3, and 4 described in Section III were met during the course of this project. To wit:

1. A generic event language called Flea (Formal Language for Expressing Assumptions) was developed in which people could state complex event patterns. The event pattern language was essentially FoL, with the addition of several built-in pattern operators for event sequences, intervals, and so on. As far as we know, ours is the first event pattern language that is fully inclusive of FoL.

2. By using our earlier work on the compilation of complex triggers in the language Ap5, we were able to build the compiler for events quite rapidly. The event compiler is based upon extensions of the basic well-formed formula compiler of Ap5, built at USC/Information Sciences Institute by the same people involved in this project. Both Ap5 and Flea are based upon the dynamic language Common Lisp – new event definitions, event patterns, and other rules can be defined on the fly with great flexibility. This was an important goal going in – because we believe that a regime that requires constant recompilation would be useless in this context.

3. We devised the SoMoS (Software Monitoring Service) software system around the Flea language. SoMoS is a client/server system, where the server contains the event reasoning engine. Clients come in three different flavors:

    a) Event Definers: These are clients that send the server definitions of new classes of events. Such definitions describe the kinds of data associated with events of that class. Events are actually modeled as a special case of relations, and clients may also define relations. Some event classes are *primitive* – they simply represent the occurrence of something in the world. Other event classes are *derived* from other previously defined event classes. The specification of the derivation is an event pattern expressed in a high level language called Formal Language for Expressing Assumptions or Flea, tailored specifically for that purpose.
    b) Event Receivers: These are clients that request the server to notify them about occurrences of specific event classes, primitive or derived.
    c) Event Emitters: These clients notify the server about occurrences of specific primitive event classes.

    The server, in effect, accepts a stream of primitive events, deduces occurrences of derived events, and reports to clients the (primitive or derived) events in which they have expressed interest. Note that there can be any number of event emitters supplying primitive event occurrences, and any number of event receivers, requesting occurrences of certain event classes, and any number of event definers sending new event definitions to the server. The APIs for each of these kinds of clients defines the total SoMoS monitoring architecture.

4. We have integrated SoMoS into a variety of different systems. The server runs on both Windows-based and UNIX-based systems. The clients can be on any platform, so long as they follow the sockets-based API for the different roles they might be playing.

In addition, we also integrated SoMoS into ModusOperandi's Catalyst engineering environment (www.modusoperandi.com). This system was an example of an "advanced" programming environment into which we were able to insert monitoring as an additional service. Catalyst's knowledge about the artifacts under development enabled more interesting detection and response metaphors than were possible in UNIX.

V SOMOS Technology Overview

Monitoring is a critical component in controlling any adaptable system. Modern software systems are required to be more adaptable than their predecessors. Hence, they will require more powerful and flexible monitoring capabilities. Such capabilities seem central to network management, law enforcement for computer crimes, education evaluation, and a host of other applications that require the ability to make decisions based upon correlations between distributed events.

As part of this DARPA EDCS project, CS3 has demonstrated monitoring technology that forms an excellent starting point to build the active network monitoring infrastructure. One of the emerging results of the EDCS Project, entitled "*Software Evolution through Automatic Monitoring*", is a system called Software Monitoring Service or ***SoMoS***. ***SoMoS*** is essentially a data fusion service, collecting low level data, combining it to produce results of interest to consumers, and delivering it to the consumers that requested it. ***SoMoS*** accepts primitive events and delivers notifications of derived events over the network and is therefore a useful service for some network applications. This system already has the ability to add monitors dynamically, an important requirement for active network monitoring.

V.1 SoMoS Architecture and Technology Details

SoMoS is currently implemented as a single program running on a single host. We aim to generalize that in significant ways after this project. The existing ***SoMoS*** architecture is illustrated in Figure 1. It has a traditional client/server flavor if one views the reasoning engine as the server. However, it is more useful just to view ***SoMoS*** as a service and the various roles that "clients" of ***SoMoS*** might play in the architecture, characterizing the different ways in which one might interact with clients:

d) Event Definers: These are clients that send the server definitions of new classes of events. Such definitions describe the kinds of data associated with events of that class. Events are actually modeled as a special case of relations, and clients may also define relations. Some event classes are *primitive* – they simply represent the occurrence of something in the world. Other event classes are *derived* from other previously defined event classes. The specification of the derivation is an event pattern expressed in a high level language called Formal Language for Expressing Assumptions or Flea, tailored specifically for that purpose.

e) Event Receivers: These are clients that request the server to notify them about occurrences of specific event classes, primitive or derived.

f) Event Emitters: These clients notify the server about occurrences of specific primitive event classes.

The server, in effect, accepts a stream of primitive events, deduces occurrences of derived events, and reports to clients the (primitive or derived) events in which they have expressed interest. Note that there can be any number of event emitters supplying primitive event occurrences, and any number of event receivers, requesting occurrences of certain event classes, and any number of event definers sending new event definitions to the server.

Flea is an event calculus based upon Ap5. New event classes (primitive or derived) are defined in Flea. At the present time, Flea supports such constructs as:

- logical operations, e.g., *and*, *or*, *not*,
- quantification: universal via *for-all*, existential via *exists*
- computational facilities, e.g., arithmetic, string comparison
- temporal facilities, e.g., *then* (one event following another), *in_time* and *too_late* (one event following or failing to follow another within a given time limit respectively), *then_excluding* (similar to then but disallowing intervening events of some other type)
- data aggregation, e.g., *max*, *sum*, *count*

A more complete language description can be obtained from: `www.compsvcs.com/flea.html`. Also, the manual for Flea is provided as part of CDRL Item A006.

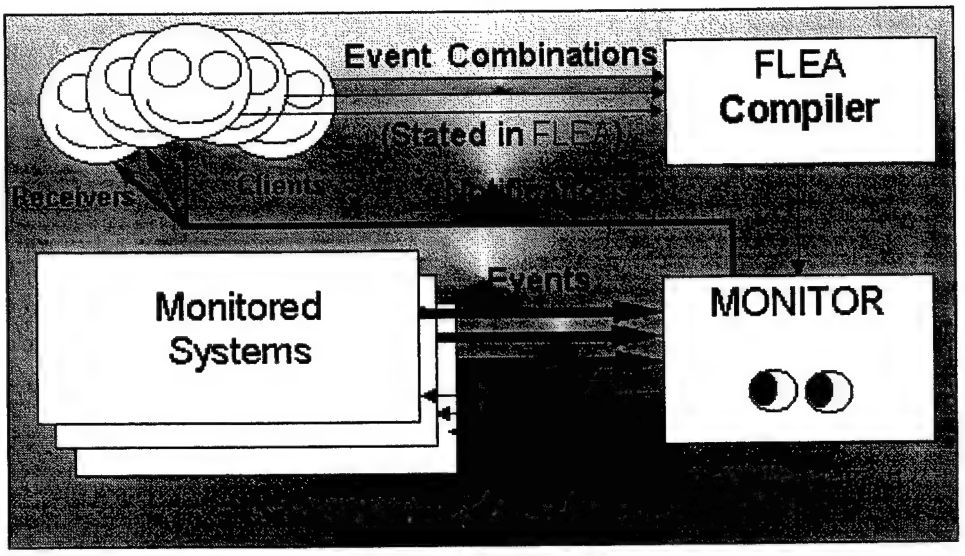

Figure 1: *SoMoS* Monitoring Architecture

SoMoS, as currently constituted, retains a complete history of primitive events that it has seen. This preserves the full generality in permitting future definition of event patterns that refer to past events (but at a cost - that of storage for all primitive events).

The event definer has the opportunity, though not the obligation, to provide "annotations", similar to "pragmas" which the server uses to make algorithmic choices in its compilation of detector code to recognize derived events. The typical annotations in Flea specify data representations for storing the primitive events and estimates of data volume. Currently the Flea compiler accepts these annotations as given and chooses algorithms that minimize the expected cost of its computations under these assumptions. That cost is simply an estimate of runtime for the server process alone.

The code output by the compiler is in at least one sense already ideally organized for distributed computation. It is internally organized as a set of nodes that send tuples to each other. These nodes can in principle be distributed across different machines on the network, as long as each has access to the data it needs in addition to the input tuples it accepts from other nodes.

To summarize, the primary qualities of ***SoMoS*** are its powerful event reasoning capabilities, dynamic (on-the-fly) extensibility, and support for users of varying levels of sophistication (from the end-user to the domain experts -- these latter being the ones who are capable of providing annotations to further help the compiler). These qualities are likely to be needed by any non-trivial monitoring activity in a dynamic environment, including any distributed decision support systems.

V.2 SoMoS Tool Overview

SoMoS is a client/server system. The server is the main reasoning component in the system. Clients play the role of event definers, emitters, or receivers as shown in Figure 1. The hardware and software requirements for SoMoS will clearly depend on whether one is interested in a SoMoS server or client. All clients use a socket-based protocol to connect to the SoMoS server. Thus, for remote clients, an Internet connection is presumed.

We have SoMoS servers that can run on both SUN Solaris and Windows NT/9X platforms. The server is distributed as a standalone program, that can be executed directly from the operating system. The server can be run from Sparc stations (if you are running UNIX), and from standard PC's that are running Windows NT or Windows 9X (i.e., 32 bit and 32 Meg of Ram Minimum). The server software uses up about 10 Meg of disk storage.

The SoMoS clients, whether they are event definers, emitters, or receivers can run on any platform. Application Program Interfaces to SoMoS for each of these roles are documented in the SoMoS User's Manual. All programs follow a common protocol to establish a connection to the SoMoS server. From that point on, there are separate APIs corresponding to each of the different kinds of roles that one might play:

a) Event Definers would use Flea definitions as published in the manual to communicate with the server.

b) Event Emitters are provided a simple syntax to generate occurrences of new events.

c) Event receivers must register initially via a socket stream as to what event they are interested in subscribing to. Subsequently, all occurrences of that type of event will be sent as individual notifications to the receiving stream. This notification has a published format that can be used by the receiving program (documented in the SoMoS manual, CDRL Item A009). The receiver can choose to do any computation or action automatically when the notification is sent to the stream.

VI Integration of SoMoS and Catalyst

A key objective of this research was to integrate monitoring into a more advanced programming environment. Our efforts with KBSA/ADM became infeasible because that system was no longer being supported by Andersen Consulting. After a few false starts with other candidates, we were directed to use Catalyst from Modus Operandi as the substitute environment by our Rome Labs sponsors. As it turned out, we were quite happy with how things worked out.

Catalyst is an engineering environment, rather than a programming environment. It provides uniform navigability for users and tools to integrate applications that involve heterogeneous databases. Catalyst is an object-oriented system, based upon the Corba standard. This gave us a new playground to try to integrate events a la SoMoS.

Concept of Integration

Catalyst servers notify the Flea Monitor of changes via relationship(s) from Catalyst objects to a Flea Gateway Object (FleaGOB). The FleaGOB then forwards the event to the Flea Monitor (See Figure 2). Figure 2 shows that the FleaGOB is not itself a Flea Monitor, but it communicates with one. (Note: Whether the FleaGOB turns out to be a monitor vs. talk to a monitor is transparent to users of the Catalyst framework, and thus is not a huge concern.)

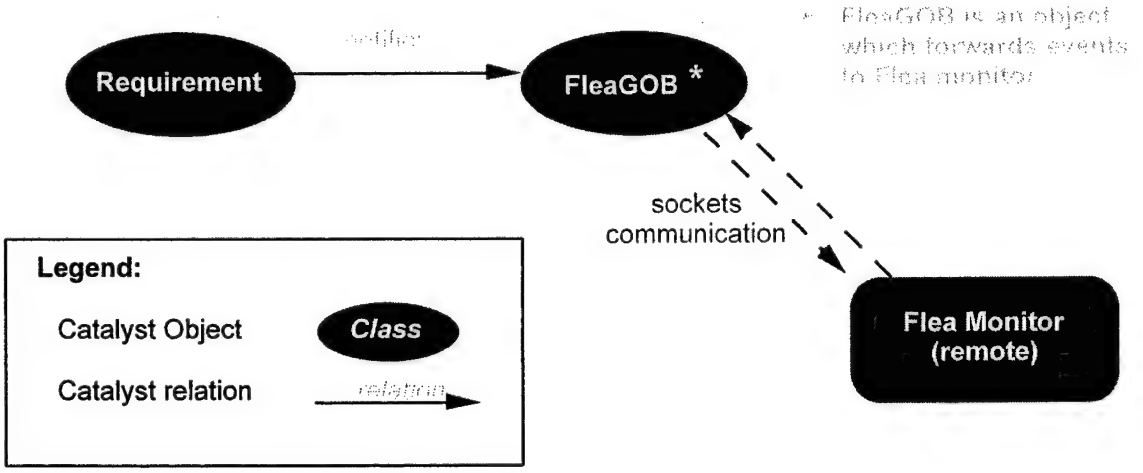

Figure 2. GOB High-level Concept

Figure 2 shows more technical details of the GOB concept. The Flea Monitor receives event notifications from related Catalyst objects and possibly other sources. It notifies the FleaGOB server of any events which the GOB server is interested in (i.e., any events which have been registered for).

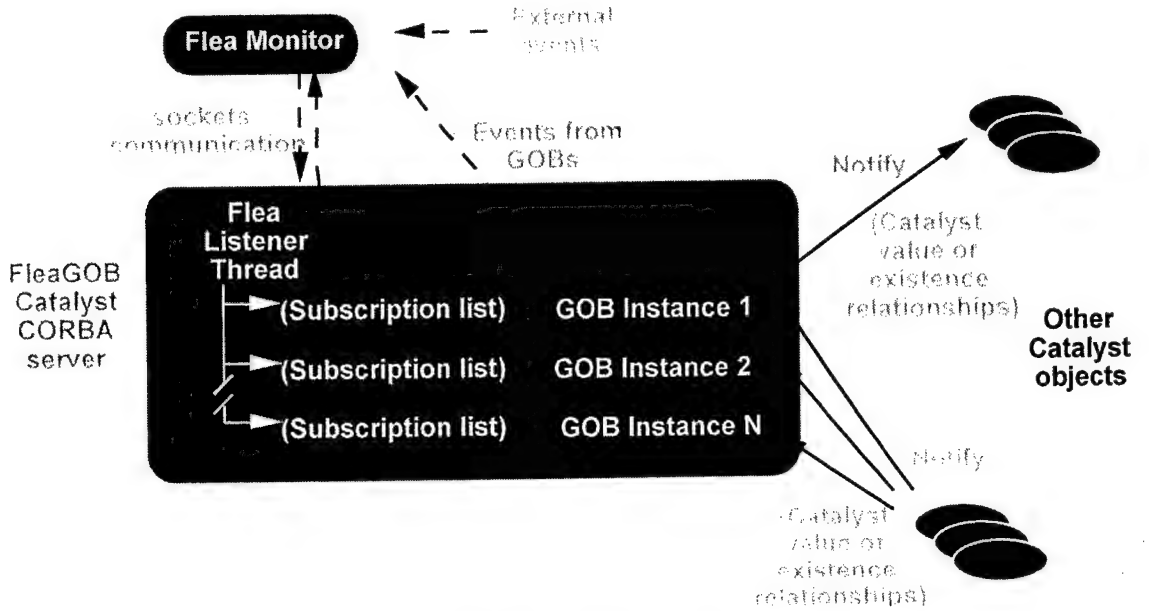

Figure 3. GOB Details

There can be multiple instances of FleaGOB objects. Each such object is notified of events of interest to it. The FleaGOBs in turn forward any Catalyst Notification Messages to related objects, based on the value or existence semantics of their relationships. Further, other Catalyst objects may notifiy a FleaGOB instance of a value change or deletion via Catalyst value or existence semantics relationships. The FleaGOB then forwards the event on to the Flea Monitor, where it can be reasoned over. The natural level of Catalyst/Flea integration would be to fully instrumenting Catalyst servers to support the notion of events—or "relations" in the Flea sense—*over all* objects, not just those related to a FleaGOB.

VII Final Project Demonstration

The final demonstration of the SoMoS/Catalyst integration was conducted at Cape Canaveral Air Station (CCAS) on January 19th, 1999. The demonstration site and scenario both reinforced the fact that SoMoS is both usable and provides the distributed event-based decision support promised during this project.

Of course, that is a more general context than the specific software evolution milieu that was planned for this project. On the other hand, what we showed was that events can be used to integrate enterprise applications together. This is the more general, and powerful, idea underlying this entire project.

Catalyst is being used at CCAS as the enterprise solution to data heterogeneity. This has led to integrated applications that can navigate seamlessly from one repository to another. However, there is no control integration. What if one wanted to recompute something in one tool when an event occurred in another? This was seen as an important problem to which there was no solution in Catalyst. SoMoS was the answer.

The demonstration scenario featured Thermal Runaway in the clean rooms at CCAS. There was already a temperature monitoring system called MetaSys that did the actual monitoring of raw temperatures. However, there was no capability to provide anything beyond threshold monitoring, and generally notifying operators. There was no general correlation capability or ability to integrate with other applications.

Through Catalyst as an intermediary, we were able to get temperature notifications sent to SoMoS as external events. In addition, there was a water valve which could be open to varying percentages to affect the clean room temperature as needed – either to cool it or to let it warm up to ambient temperature.

There were thus two external events in the demonstration scenario (recall that the first parameter for all events is invariably a timestamp – the time at which the event occurred):

TEMP event captures the temperature of a building, the second parameter, at a particular time:

(defevent TEMP :external (timestamp string number))

OPEN event describes the percentage to which the water valve was opened in a building (the second parameter, as with TEMP):

(defevent OPEN :external (timestamp string number))

As temperatures change over time, and the water valve is adjusted to varying percentages in response, these events are forwarded to SoMoS.

Typical of the kinds of analysis that can be conducted are as follows:

```
(OPEN 12345.678 Bldg3 80)      <-- Cold water valve for building 3 is 80% open
(TEMP 12345.678 Bldg3 75.0)    <-- temp for same building at 75.0 degrees
(TEMP 12445.678 Bldg3 76.2)    <-- temp rising 100 seconds later
(TEMP 12545.678 Bldg3 77.4)    <-- temp still rising 100 seconds later
(OPEN 12545.678 Bldg3 90)      <-- valve open to 90% now
(TEMP 12645.678 Bldg3 77.6)    <-- temp still rising, but slower
(OPEN 12745.678 Bldg3 100)     <-- valve open to 100% now -- max flow!
(TEMP 12845.678 Bldg3 77.7)    <-- temp still rising
(TEMP 12945.678 Bldg3 77.8)    <-- temp still rising
```

The example trace above shows that it is NOT from individual events that one generally deduces problems. Trends and correlations are much more important in rendering decisions. The fact that the temperature is rising at the same time that the valve is 100% open is actually the critical condition that must be detected. While any dedicated sensing software can be hard-wired to detect conditions such as the above, it cannot handle new conditions dynamically as SoMoS can.

Call the "dangerous condition" above an open-rising event (the valve is fully open, but the temperature is still rising). This is described in Flea as below:

```
(defevent open-rising :definition ((time bldg) s.t.
        (seq (open $ bldg 100) (rising time bldg) :without (open $ bldg $))))
```

Such an event definition can be sent to SoMoS on the fly. At this point, new detector code is compiled to flag the new derived event, which is essentially a complex pattern of existing external events. Anyone interested in the above event would use the SoMoS API to register as a receiver of "open-rising" events, and would be notified automatically when that pattern of events is detected.

We also looked at related trend analysis of temperature and other events. For example, we might be interested in the fact that the temperature is now higher or lower than ever in recorded history - meaning we have no relevant experience on which to draw. That might be very crucial in how we handle the situation or, more likely, how much we panic!

In fact, we could even label extreme episodes, e.g., the heat wave of '97 and have the monitor tell us when we're outside the bounds of experience OTHER than those episodes, and also tell us which episodes we are still inside of. So, now we imagine our monitor saying something like "Golly, it ain't been this hot since the heat wave of '97!"

More generally, we might expect on theoretical grounds that the second derivative of TEMP (which captures whether the rate at which the temperature is rising or falling) should depend inversely on OPEN. We could, first, compute the second derivative, and second announce when we go outside the envelope of recorded history in the following sense:

Call the 2'nd derivative of temperature at time t d2TEMP(t) and the percent open at time t OPEN(t). If there have been 4 previous times with data that surround the data as of now, i.e.,

OPEN(t1) >= OPEN(now) and d2TEMP(t1) >= d2TEMP(now),
OPEN(t2) >= OPEN(now) and d2TEMP(t2) <= d2TEMP(now),
OPEN(t3) <= OPEN(now) and d2TEMP(t3) >= d2TEMP(now),
OPEN(t4) <= OPEN(now) and d2TEMP(t1) <= d2TEMP(now),

then "now" is within the envelope of previous experience. If not, "now" is not within the envelope of experience, and appropriate responses may be taken.

If we're outside that envelope then there's something unusual going on, such as (1) a fire, (2) an obstruction in the pipe, (3) a record outside temperature – all of these are truly extraordinary circumstances.

Once again, we could label the genuine exceptional events, and report something like "the last time the temperature accelerated upward this fast with the pipe 100% open was when the boiler exploded!"

(defevent record-high :definition
((x y z) s.t. (and (temp x y z)
(not (e (xx zz) (and (temp xx y zz)(< xx x)(> zz z)))))))

Another suggested analysis to complement the basic framework was to tag "extreme" events. As an example, we could invent the relation

(defrel extreme-event :types (description start-time end-time))

Each extreme event is characterized by a description – a formula. For example, the fire in the lab, which accounts for the extreme temperatures between time t1 and t2. Then we'd probably want an event like "record-normal-high" which has a signature of (timestamp place temperature). Record-Normal-High is defined similar to above but we exclude previous times between the start and end of an extreme-event, and when we reach a record-normal-high then we might recount the extreme events in which higher temperatures were recorded.

;; the times that fall in extreme events

```
(defrel in-extreme :definition
        ((x) s.t. (e (t1 t2) (and (extreme-event $ t1 t2) (<= t1 x) (<= x t2)))))

(defevent record-normal-high :definition
        ((x y z) s.t. (and (temp x y z) (not (in-extreme x))
                           (not (e (xx zz) (and (temp xx y zz)(< xx x)(> zz z)
                           (not (in-extreme xx)))))))
```

These types of analysis can be coded in Flea and sent to SoMoS. We were able to demonstrate much of this during the CCAS demonstration. More importantly, we could show that new applications could be written that responded to these new events just as if they had always been known about. This incremental and dynamic feature of SoMoS makes it more useful than its competitors.

VIII Lessons Learned

As with most research projects, we set out with one set of plans, and ended up refining them as we went along. However, the journey has proved interesting and fruitful, and set this company up for a productive future in the monitoring arena. Lessons learned were as follows:

a) When we started this project, we believed we could steer clear of the thorny problems of instrumenting software systems to emit events. We wanted to finesse this problem by providing an interface. However, we discovered that while the correlation problem is very difficult, so is the instrumenting problem. In other words, getting access to the raw event stream is just as difficult as correlating them. To this problem we provided little by way of solution, and it kept re-occurring.

b) The use of Common Lisp as a dynamic language platform upon which to host Flea was not a difficult decision for us, given this company's historical predilections. It was, however, still startling how easily we were able to achieve incremental, dynamic event specification pretty much for free. The future would seem to lie in dynamic languages such as Lisp, Java, and so on.

c) Integration with basic operating systems was not very difficult. We were pleasantly surprised with the universal use of sockets as the underlying communication mechanism between the SoMoS server and clients – regardless of whether the communication was local, over a LAN, or over the Internet.

d) If we had done this over, we probably would integrate monitoring at an even lower level than we managed to. Our events are application-generated. These are most meaningful, but hard to come by.

e) We learned that while many tools advertise the format of data they consume and produce, few, if any, advertise the way they signal things going on within them. In other words, encapsulation of software still does not seem to encompass the events necessary to pull together integrated applications.

f) Many systems, particularly those in the real-time domain, seemed to have efficiency and volume concerns that we were not able to adequately address in the client/server architecture. Such environments require much more efficient, compiled in detector code which was beyond the scope of the envisioned work.

IX Commercialization Successes and Future Potential

This project has resulted in SoMoS, now a licensed product of Cs3. Monitoring is a horizontal technology of increasing importance in today's distributed computing environments. We have

found tremendous commercial opportunities for SoMoS during the course of this project. Some of these have already been consolidated, others remain to be locked up. To summarize:

a) Motorola was the first customer to buy SoMoS, based upon our EDCS 1997 demonstration. They were impressed not only by its functionality but by its design as a stand-alone generic monitoring service, quite separate from the systems being monitored. Motorola is interested in using monitoring technologies to ensure continued enforcement of their design policies in cellular networks even as customers modified their software systems. Delivery was made in September 1998. This was our first commercial sale of SoMoS.

   A pilot project has been started to evaluate how well SoMoS is doing in this application. Evaluation is pending.

b) Cost Management Solutions of San Diego, CA approached Cs3 to build distributed software support for a process called Aim&Drive that they teach on supply chain cost management. The process is intended for large buyer companies and organizations to build strategies to control costs in a supply chain with their supplier companies. CMS has been enormously successful in teaching this process to large Fortune 500 companies. However, it became clear that they needed sophisticated software to support this process.

   Aim&Drive software would record the strategies, browse the strategies, and, most important, provide technology to allow people to collaboratively develop such strategies without the need for costly and inefficient physical meetings.

   Event monitoring is used as the technical basis for building a framework for distributed collaboration. Various individuals are able to cooperatively develop cost management strategies without leaving their offices. The key to this is the ability for everyone to know what is going on at the workstations of others. We have implemented this as a sophisticated set of events on top of SoMoS. Every client activity initiates events, which are then processed by the server, and sent to the subscribing receivers.

   The Beta Release of this software was in February 1999. We expect the full product to be released in June 1999. This product is being developed as a joint venture with Cost Management Solutions, and it is a derivative of SoMoS.

c) Cs3 has been awarded an SBIR grant from Darpa entitled "An Event Monitoring Framework for Automatic Network Diagnosis". This project will look at generalization of SoMoS to a federated, rather than client/server, architecture. This will provide many application possibilities in networking – a very lucrative commercial area.

d) Cs3 is also pursuing other areas of application via collaboration links with partners in education, enterprise computing, and other areas that can utilize sophisticated monitoring.

DISTRIBUTION LIST

addresses	number of copies
DEBORAH A. CERINO 525 BROOKS RD ROME NY 13441-4505	15
COMPUTING SERVICES SPT SOLUTIONS 5777 W. CENTURY BLVD SUITE 1185 LOS ANGELES CA 90045-5600	5
AFRL/IFOIL TECHNICAL LIBRARY 26 ELECTRONIC PKY ROME NY 13441-4514	1
ATTENTION: DTIC-OCC DEFENSE TECHNICAL INFO CENTER 8725 JOHN J. KINGMAN ROAD, STE 0944 FT. BELVOIR, VA 22060-6218	2
DEFENSE ADVANCED RESEARCH PROJECTS AGENCY 3701 NORTH FAIRFAX DRIVE ARLINGTON VA 22203-1714	1
ATTN: NAN PFRIMMER IIT RESEARCH INSTITUTE 201 MILL ST. ROME, NY 13440	1
AFIT ACADEMIC LIBRARY AFIT/LDR, 2950 P.STREET AREA B, BLDG 642 WRIGHT-PATTERSON AFB OH 45433-7765	1
AFRL/HESC-TDC 2698 G STREET, BLDG 190 WRIGHT-PATTERSON AFB OH 45433-7604	1

ATTN: SMDC IM PL
US ARMY SPACE & MISSILE DEF CMD
P.O. BOX 1500
HUNTSVILLE AL 35807-3801 1

COMMANDER, CODE 4TL000D
TECHNICAL LIBRARY, NAWC-WD
1 ADMINISTRATION CIRCLE
CHINA LAKE CA 93555-6100 1

CDR, US ARMY AVIATION & MISSILE CMD
REDSTONE SCIENTIFIC INFORMATION CTR
ATTN: AMSAM-RD-OB-R, (DOCUMENTS)
REDSTONE ARSENAL AL 35898-5000 2

REPORT LIBRARY
MS P364
LOS ALAMOS NATIONAL LABORATORY
LOS ALAMOS NM 87545 1

ATTN: D'BORAH HART
AVIATION BRANCH SVC 122.10
FOB10A, RM 931
800 INDEPENDENCE AVE, SW
WASHINGTON DC 20591 1

AFIWC/MSY
102 HALL BLVD, STE 315
SAN ANTONIO TX 78243-7016 1

ATTN: KAROLA M. YOURISON
SOFTWARE ENGINEERING INSTITUTE
4500 FIFTH AVENUE
PITTSBURGH PA 15213 1

USAF/AIR FORCE RESEARCH LABORATORY
AFRL/VSOSA(LIBRARY-BLDG 1103)
5 WRIGHT DRIVE
HANSCOM AFB MA 01731-3004 1

ATTN: EILEEN LADUKE/D460
MITRE CORPORATION
202 BURLINGTON RD
BEDFORD MA 01730 1

OUSD(P)/DTSA/DUTD 1
ATTN: PATRICK G. SULLIVAN, JR.
400 ARMY NAVY DRIVE
SUITE 300
ARLINGTON VA 22202